Study and Implementation of Single Trace Attack against RSA key generation in Intel SGX SSL

Maharishi Bhargava, Vamshi Chiluka, Diptyaroop Maji

CS 741 Course Project

Instructor: Professor Bernard Menezes

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

April 20, 2019

Abstract

Controlled channel attacks can be carried even on Intel Software Guard Extensions (SGX) supported environments to extract sensitive information. It takes advantage of vulnerabilities in existing code libraries and the fact that the underlying Operating Systems (OS) manages the mapping between virtual and physical pages for all processes, including processes executed inside hardware enclaves. In the RSA key generation procedure of Intel SGX SSL, which uses OpenSSL library, binary euclidean algorithm (BEA) is used to validate the RSA key parameters. These RSA key parameters are generated within SGX Enclave only. We can launch a controlled channel attack on binary euclidean algorithm to identify most of the bits of the generated prime factors p,q. BEA algorithm is used to test whether $p-1$ and $q-1$ are co prime to public exponent e or not. Based on the pages accessed by the BEA running in enclave, we can obtain all the bits, except 16(if e is 65537) of one of the two prime factors of public modulus (remaining 16 bits can be recovered based on related observations) and find the other prime factor. After finding the two prime factors, the private key can be found. In this project, we intend to study and implement the side channel attack against RSA key generation in Intel SGX SSL, as described in [1].

Contents

List of Figures

1. Introduction

Intel Software Guard Extensions (Intel SGX) is a technology which is built into modern Intel processors. It aims to provide confidentiality & integrity of code and data, even if privileged software like underlying OS is malicious. SGX Developers can partition security-critical information into enclaves, which are private regions of memory. Contents in an enclave are protected from other processes and the OS. An untrusted OS cannot directly read or write in enclave memory. OS manages the mapping between virtual and physical pages for all processes, including processes executed inside hardware enclaves. Security guarantees of SGX make it difficult to perform any direct attack in SGX. However, controlled channel attacks can still be carried on in such environments to extract sensitive information. Controlled channel attacks on page table of program executing in enclaves, reveal page-level access patterns of the program.

Side channel attacks are often performed on secret key operations such as decryption and signature generation of digital signature schemes. They usually require multiple observations of memory accesses in multiple executions to break an implementation. Hence, one time operations like key generation routines were considered out of scope for such attacks. However, with the introduction of shielded execution environments like Intel SGX, we can recover the secret keys just by observing memory access pattern of a single execution.

In the RSA key generation procedure of Intel SGX SSL, which uses OpenSSL library; binary euclidean algorithm (BEA) is used to validate the RSA key parameters. BEA algorithm is used to test whether p-1 and q-1 are co-prime to public exponent e or not. The RSA key parameters are generated within an SGX Enclave. Since, the operating system is responsible for managing virtual to physical mappings of SGX enclaves, it can record page accessed by the enclave. We can launch a controlled channel attack on binary euclidean algorithm to identify most of the bits of the generated prime factors p,q. Based on the pages accessed by the BEA running in enclave, we can establish linear equations on the secret parameters. We can obtain all the bits, except 16 of one of the two prime factors of public modulus. Remaining 16 bits can be recovered based on related observations.

In this project, we intend to study and implement the side channel attack against RSA key generation in Intel SGX SSL, as described by Weiser et. al. [1]. This attack can recover the secret key with minimum computational effort on a commodity PC, in less than 12 secs. In Section 2, we discuss the vulnerability in the RSA key generation procedure of Intel SGX SSL. In Section 3, we present the attack and the implementation for key recovery. In Section 4, we discuss evaluate the implementation of the attack. In section 5, we present personal opinion and suggestions. Finally, we conclude in section 6.

2. Vulnerability

To exploit the vulnerability we consider an enclave that is generating RSA key dynamically. The OS on which the enclave is running is untrusted and compromised. To generate RSA key, first public exponent e is assigned value 65537. Then RSA_generate_key_ex function of openssl is called, in this pointer to RSA object, pointer to public exponent (BIGNUM) object, key size and pointer to a callback function is passed. While generating RSA key, p and q are generated using BIGNUM_PRIME_ function and then BN_gcd function is called to check if p-1 and e are co-prime. Similarly BN_gcd function is called once again to check if q-1 and e are co-prime. In openssl-1.1.0j from BN_gcd, euclid function called. In euclid function, the binary gcd euclid algorithm is defined which is used to calculate gcd of two numbers.

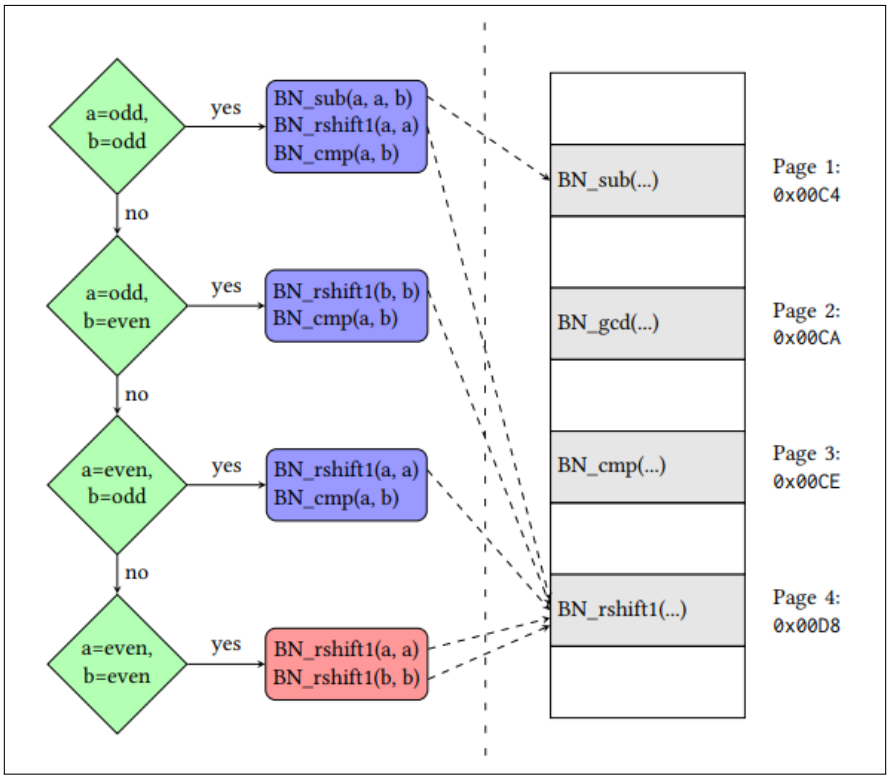

Figure 2.1: Control flow of binary GCD algorithm[1]

In our case, initially the value of a will be equal to $p-1$ or $q-1$ and b will be equal to b that is 65537. Depending on the value of a and b sequence of functions are executed as shown figure **??**.Each of the functions BN_sub, BN_rshift and BN_cmp are present on different physical pages, also the BN_gcd function is on different page.

[1]Weiser, S., Spreitzer, R. and Bodner, L., 2018, May. Single trace attack against RSA key generation in Intel SGX SSL. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (pp. 575-586). ACM.

Let's assume that BN_sub is on page 1,BN_gcd in on page 2, BN_cmp in one page 3 and BN_rshift is on page 4.
Let's take an example where $p = 11083$ and $e = 17$. Therefore $a = 11082$ and $b = 17$. The sequence of functions executed and pages accessed are shown in table 2.1. In the table P1 means page 1, P2 means page 2,P3 means page 3 and P4 means page 4.

a	b	**Page Access Sequence**	**Performed Operation**
11082	17	P4,P2,P3,P2	$a_{i+1} = a_i/2$
5541	17	P1,P2,P4,P2,P3,P2	$a_{i+1} = (a_i - b_i)/2$
2726	17	P4,P2,P3,P2	$a_{i+1} = a_i/2$
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.

Table 2.1: Page Access Sequence

As the initial value of a is even and value of b is odd, the page access sequence "P4,P2,P3,P2". In the next iteration , value of a is odd and value of b is also odd, we get the page access sequence is "P1,P2,P4,P2,P3,P2". We get one of these two sequences till the value of a is greater than b. After that we can get other sequences as well but we are not interested in them.
By making pages non executable, the malicious OS can actually get the pages which were accessed while finding the GCD. As page 3 followed by page 2 is accessed in every iteration we can ignore it. The remaining part of page access sequence can be used to launch the attack.

3. Attack

In our implementation of the attack, which closely follows the attack carried out by the authors [8], there are three main stages:

- Carrying out the attack in an SGX environment, where the victim code runs inside an enclave.
- Generate a trace from the pages accessed by the victim function while it calculated the GCD.
- From the trace and using publicly available (e, n), formulate equation to find $(p - 1)$.

Once p is found, generating q and d are trivial. (Note that we use p and q interchangeably without any loss of generality).

3.1 Running attacker and victime inside SGX

SGX programs usually include 2 key parts — functions having sensitive data are written in Enclave.cpp whereas all other parts of the code is written in App.cpp. Functions written in Enclave.cpp are encrypted and executed from inside an enclave (provided by SGX), and are not visible to other processes or even the OS. Such functions can be called using ECALL, while coming out of the enclave to do some work requires an OCALL. Our setup has the attacker code running within App.cpp and the victim code calling $RSA\_generate\_key\_ex()$ in Enclave.cpp. The attacker code calls the victim enclave via $ecall\_generate\_RSA\_key()$. As discussed in Section 2, we know the pages that the GCD function will access, and hence, we make those pages non-executable beforehand. To obtain information about the victim enclave and make pages executable/non-executable from our attack app in userspace, we use SGX-Step [2], and the libsgxstep [3] library provided by it. We use *objdump* to obtain the respective page offsets. For instance, to find the page offset of the page having $BN\_sub()$, we use the following command:

```
objdump -t enclave.so | grep BN_sub
```

We have also written a page fault handler which is triggered whenever the victim code tries to call any functions, but faults due to lack of permission. We know that the first call will always be to BN_gcd, so initially, we make only the page containing BN_gcd as non-executable. In general, when a page is faulted, we note which function is called (if the page containing BN_gcd is faulted, we add g to our initial trace file. Similarly, we add r for $BN\_rshift1()$ and s for $BN\_sub()$). We then make that page executable for the time being so that the flow of execution can process, while we make the previously faulted page (say, p_page) non-executable

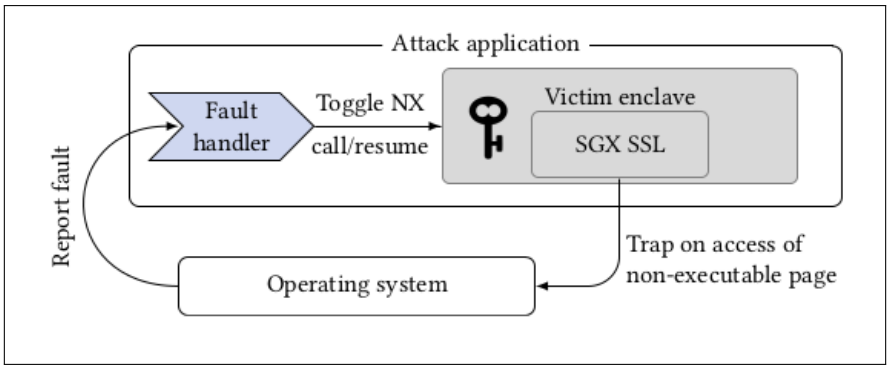

Figure 3.1: Basic attack priciple[1]

again so that it page fault is triggered again the next time the victim function tries to access *p_page*. Figure 3.1 show the the attack in a nutshell.

Once everything is set up, the *ecall_generate_RSA_key*() is called, and the page fault pattern in noted down in a file *page_faults.txt*, which is used to generate the required pattern. This is explained in the following section.

3.2 Trace generation

The *page_faults.txt* file contains data as shown in figure 3.2. Each line of the file ends with a "g". This file contains information about the page accessed during the calculation of the gcd as well as some other functions. We need to extract the data which was generated during the calculation of gcd.

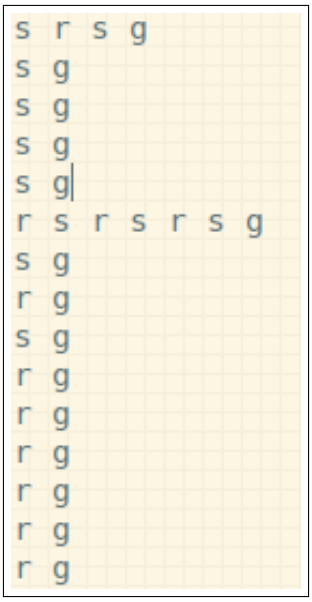

Figure 3.2: Excerpt from *page_faults.txt*

From our observation we found that there are only two blocks of data in which "r g" is present atleast $keysize/2$ times and also it contains "s g" followed by "r g" or only "r g" sequence. No other sequence(for e.g. "s r s g") should be present in that block.

[1]Weiser, S., Spreitzer, R. and Bodner, L., 2018, May. Single trace attack against RSA key generation in Intel SGX SSL. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (pp. 575-586). ACM.

Using a python script, we find such a block from the bottom of file. The block may contain "r g" sequence more than $keysize/2$ times due to some other functions. We extract the first part of the block which contains "r g" sequence exactly $keysize/2$ times. This extracted sequence is written to a file called *trace.txt*, which is used to perform further part of attack.

3.3 Backtracking to find p and q

The generated trace can be used to identify the branches that were accessed in the binary gcd function. We have a python script which will take the generated trace as input and outputs the unknown parameters of RSA p and q. The script reads the trace and first, identifies the branches that were accessed in gcd. It then uses the corresponding branch equations and obtains the unknown secret a using backtracking.

Whenever, the trace consists a continuous access to s, g, r, g it means branch1 was accessed, and when the trace contains r, g, it means branch3 was accessed. Consider the set of linear equations for branches (Equations 3.1, 3.2, 3.3, 3.4 in terms of i). We iteratively add equations by incrementing i by one in every iteration. The algorithm terminates after n steps, when $a_n = gcd(a, b)$ and $b_n = 0$.

$$a_{i+1} = \frac{a_i - b_i}{2} \tag{3.1}$$

$$b_{i+1} = \frac{b_i}{2} \tag{3.2}$$

$$a_{i+1} = \frac{a_i}{2} \tag{3.3}$$

$$a_{i+1} = \frac{a_i}{2}, b_{i+1} = \frac{b_i}{2} \tag{3.4}$$

With recursive substitution, the initial unknown a can be expressed in terms of a_n, b_n, i.e.,

$$a = f(a_n, b_n) \tag{3.5}$$

By choosing m^{th} iteration, where the two variables am, bm are swapped for the first time, the value of m is approximately $KeySize - log(e)$. Until m^{th} iteration, the value of b has not changed ($b_m = b$) . The initial unknown a can be expressed in terms of a_m, b_m, i.e., $a = f(a_m, b)$, or more precisely:

$$a = a_m * c_a + b * c_b \tag{3.6}$$

where constants c_a and c_b are known, while a and a_m are unknown.

The additional information known after m^{th} iteration, that $a_m < b$, implies that a_m is less than e. We perform a brute force using all the values of a_m in range $[1, e)$, and evaluating the corresponding value of a using above equation. For each candidate a, we compute $GCD(a + 1, N)$, here $a + 1$ is same as p. When $GCD(a + 1, N) > 1$, we have recovered the correct value of a which is same as $p - 1$. Hence, both the secret parameters p and q can be computed, where $q = N/p$. The decryption key d can be computed using:

$$\phi(n) = (p - 1)(q - 1),\ d = e^{-1} mod\ \phi(n) \tag{3.7}$$

4. Evaluation

We run our experiments using the following setup:

- Intel i7-8750H CPU @ 2.20GHz
- 16 GB memory
- Intel SGX v2.4
- Sgx-step (compatible with SGX v2.4)
- Intel SGX-SSL (OpenSSL 1.1.0j) — *rsa_gen.c* file modified to source before patch

We tested our code on modulus(n) sizes varying from 128 *bits* to 8192 *bits*, while keeping the value of e fixed to 65537. By OpenSSL [4] convention, size of p and q is half the size of n. Figure 4.1 shows the time taken by the attack to generate p and q starting from the initial page fault pattern obtained. We do not consider the key generation time in our evaluation, as the prime numbers generated by the algorithm may not be co-prime to e, and so, the algorithm may generate it multiple times, leading to inconsistency in results. From the figure, it can be seen clearly that

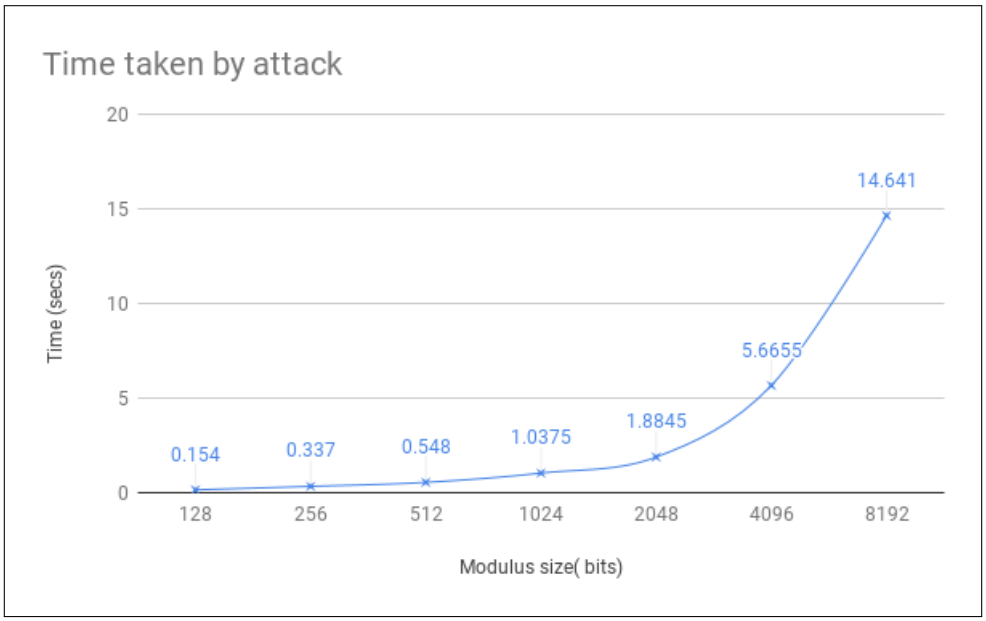

Figure 4.1: Time taken by attack to find p

initially time taken increases linearly with modulus size and although it increases exponentially when we go beyond 1024 *bits* modulus size, the time taken is still much less when compared to other side channel attacks (to find 8192 *bits* modulus, this attack takes only 14.641 secs on our system).

4.1 Attack Time Complexity

In each iteration of the GCD algorithm, the value of a ($= (p-1)$, given as one of the inputs to the GCD algorithm) reduces by at least half, i.e., in each iteration, a decreases by 1 bit. So, maximum number of iterations will be the size of the modulus. Also, we are considering till $a > e$. Hence, the number of iterations is bounded by $|modulus| - log_2(e)$. For example, in case of 4096 bit modulus, maximum number of iterations will be $4096 - 16 = 4080$. Thus, it can be done in $O(1)$.

Once the value of a becomes less than e, we apply brute force technique to find the original value of a (equation 3.6). As $e = 65537$, $a_m \epsilon [1, 65537)$ in equation 3.6. Again, this can be computed in $O(1)$.

Hence, the time complexity to recover p is $O(1)$.

5. Miscellaneous

This section discusses the challenges that we faced during our project and also suggests some possible extensions.

5.1 Challanges Faced:

Compatibility issues, while setting up and building SGX, SGX-STEP, INTEL-SGX-SSL (OPENSSL)

We faced several compatibility issues like compilation errors, deprecated API errors, different compiler versions etc., while biulding the aforementioned components as the attack is about an year old and the libraries have been upgraded. To solve this, we took the existing makefile given in SGX SampleCode and from the authors' source code [8], studied the authors' makefile in detail, and combined and modified them extensively to suit our needs.

Reverting OPENSSL to unpatched version:

Current version of openssl uses BN_mod_inverse to find the GCD of two numbers. For this attack he had to go back to the unpatch version which used BN_gcd function to find the GCD. Directly going back to the previous version using git was not useful as it had other compatibility issues. Therefore our first task was to understand the RSA_generate_key_ex function and then find the code where the GCD of p,e and q,e is calculated. Then we replaced the existing BN_mod_inverse with BN_gcd and made some little changes so that things work correctly (for example, we compiled rsa_gen.c file separately and added it to the existing archived sgxssl library while reverting back to the unpatched version).

Trace generation from page faults:

We decided to only make pages containing sub, rshift1 and gcd functions non-executable. The problem was these pages are not only used by BN_gcd function but are used many times by for other calculations, specially sub and rshift1. The *page_fault.txt*file contained around 10 thousand lines for 2048 bit key. The useful data for attack was only about 3 thousand lines and rest was not used in calculation for GCD. These 3 thousand lines contained data for GCD of p,e and q,e. As we used only the latter one for the attack, the task was to extract 1.5 thousand lines of data from 10 thousand. For that we had to run the code multiple time for different key sizes ranging from 16 bit to 8192 bit. Then we had to study the pattern which was created in the *page_fault.txt* file and after some observations we found the pattern. Finally we wrote a python script to extract the needed trace from the *page_fault.txt* file.

5.2 Possible Extensions

Single-Stepping Feature of SGX-Step Library

While reading about the SGX-Step Library which is used by us to mark the pages as non-executable, we came across a feature called single step feature. With the help of this feature, malicious OS can interrupt the victim enclave after every single instruction using the APIC timer. The attacker can launch enclave preemption attacks and read L1 cache data for accurately.

Hosting Attacker and Victim on Different Processes

In our attack we hosted the Attacker code in the app.cpp file of sgx and enclave code was also called from that file. As a extension we can run the attacker code as a different process. Single-Stepping feature can be used to preempt out the victim after every instruction. Due to the fair scheduling policy, the attacker code is executed, the attacker find's the page fault and write it in a trace file. This file can be later used to launch the attack and find the private key.

6. Conclusion

Even in secure environments like Intel SGX, controlled side channel attacks can be performed due to vulnerabilities in the existing code libraries and the fact the although the OS cannot see what is executed in a process inside an enclave, a malicious OS can still track the pages accessed by a process. In this project, we have successfully studied implemented a paper which takes advantage of the $BN_gcd()$ in the Intel SGX-SSL library (which uses OpenSSL), which calls different function sequences based on whether the inputs to the function are (even, even), (even, odd), (odd, even) or (odd, odd). As these functions are on separate pages, a maliciuos OS can track this from the page access pattren. Also, as during RSA key generation, one of the input to this function is $(p-1)$, p can eventually be found out by tracking these page accesses. This attack finds p and hence the private key pair (d, n) of RSA algorithm in less than 20 secs in a single trace for a 8192 bit modulus using the above technique.

This vulnerability has been patched in OpenSSL in commit $8db7946e$, and this attack now won't succeed in recent versions of OpenSSL (v1.1.2). However, it still remains an interesting attack to study and implements, and there are significant concerns that even though SGX promises to guard even against malicious OSs, there are other similar vulnerabilities in existing code libraries, which can be exploited to a good extent, and the developers should take extra care while writing the libraries to prevent similar side channel attacks and ensure security.

Acknowledgement

We thank Professor Bernard Menezes for inspiring us to find this paper and implementing it. We would like to thank Kantikumar Lahoti for helping us with setting up SGX.

We also express our gratitude to Samuel Weiser, one of the authors of this paper, whom we contacted during this project. He solved some of our queries and also shared their original source code, which helped us understand the attack in a better way. We also took some pointers from their code to implement part 1 of our attack (Section 3.1).

Bibliography

[1] Weiser, S., Spreitzer, R. and Bodner, L., 2018, May. Single trace attack against RSA key generation in Intel SGX SSL. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (pp. 575-586). ACM.

[2] Van Bulck, J., Piessens, F. and Strackx, R., 2017, October. SGX-Step: A practical attack framework for precise enclave execution control. In Proceedings of the 2nd Workshop on System Software for Trusted Execution (p. 4). ACM.

[3] Sgx-step github page: https://github.com/jovanbulck/sgx-step

[4] OpenSSL Software Foundation. 2017. OpenSSL âĂŞ Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. (2017).

[5] Intel Corporation. 2017. Intel Software Guard Extensions Developer Guide. https://software.intel.com/en-us/sgx-sdk/documentation. (2017).

[6] Intel SGX SDK/PSU github page: https://github.com/intel/linux-sgx

[7] Intel SGX driver github page: https://github.com/intel/linux-sgx-driver

[8] Single trace attack code: https://cloud.tugraz.at/index.php/s/tfiytmfBSc2KDZW

7. Appendix

Our code can be found in the following Github page:

`https://github.com/diptyaroop/RSA_attack_CS-741-project.git`

Please refer to the Readme file in the above link for detailed instructions on how to mount this attack on SGX enabled systems.

Following are some of the excerpts from our code, given here for clarification. Details can be found in the above link.

Page fault handling code:

```
static void pageFaultHandler(int signal, siginfo_t * siginfo, void *context)
{
    if(signal!=SIGSEGV)
        return;
    if(!prevFaultedPage || firstIteration)
    {
        firstIteration = false;
//      *subPage = MARK_NON_EXECUTABLE(*subPage);
        *rShift1Page = MARK_NON_EXECUTABLE(*rShift1Page);
    }
    else
    {
        *prevFaultedPage = MARK_NON_EXECUTABLE(*prevFaultedPage);
    }
    uint64_t faultPageIndex = PT_INDEX((uint64_t) siginfo->si_addr);
    if(faultPageIndex == PT_INDEX(gcdPageOffset))
    {
        printf("g\n");
        *gcdPage = MARK_EXECUTABLE(*gcdPage);
        if(prevFaultedPage == rShift1Page)
         {
         iterations++;
         *subPage = MARK_NON_EXECUTABLE(*subPage);
         }
        prevFaultedPage = gcdPage;
    }
    else if(faultPageIndex == PT_INDEX(subPageOffset))
    {
        printf("s ");
        *subPage = MARK_EXECUTABLE(*subPage);
        prevFaultedPage = subPage;
    }
```

```
    else if(faultPageIndex == PT_INDEX(rShift1PageOffset))
    {
        printf("r ");
        *rShift1Page = MARK_EXECUTABLE(*rShift1Page);
    prevFaultedPage = rShift1Page;
    }
}
```

Extracting GCD trace from page faults:

```
for i in reversed(range(len(lines)-2)):
    if(lines[i] != "r g" and lines[i]!= "s g"):
        if flag:
            break
        end = i-1
        count=0
        continue
    if(lines[i] == "s g" and lines[i+1] != "r g"):
        if flag:
            break
        end = i-1
        count=0
        continue
    if(lines[i]=="r g"):
        count+=1
    #print(i,lines[i],count)
    if(count==window_size):
        if(lines[i]!="r g" or lines[end]!="r g"):
            end = i-1
        else:
            flag=True
```

Generating equation to find a from page access pattern:

```
for i in range(swap_iteration+2):
    if(branch_trace[i]=="branch1"):
        a_den = a_den * 2
        b_num = b_num + b_den
        b_den = b_den * 2
    elif(branch_trace[i]=="branch3"):
        a_den = a_den*2
        if(b_num!=0):
            b_den = b_den*2
```